Week 1: Wednesday, Aug 22

Logistics

We will go over the syllabus in the first class, but in general you should look at the class web page at

  http://www.cs.cornell.edu/~bindel/class/cs6210-f12/

The web page is your source for the syllabus, homework, lecture notes, and course announcements. For materials that are private to the class (e.g. grades and solutions), we will use the Course Management System:

  http://cms.csuglab.cornell.edu/web/guest/

If you are taking the class for credit – or even if you think you may take the class for credit – please sign the list and include your Cornell netid. That way, I can add you to the Course Management System.

Course Overview

CS 6210 is a graduate-level introduction to numerical linear algebra. We will be studying how to solve linear systems, least squares problems, eigenvalue problems, and sinular value problems. Organized by topics, the course looks something like this:

1. Background: linear algebra, memory and performance, error analysis

2. Linear systems and Gaussian elimination; structured linear systems

3. Least squares problems

4. Unsymmetric eigenvalue problems

5. Symmetric eigenvalue problems

6. Iterative methods for linear systems and eigenvalue problems

We could also organize the course by four over-arching themes (4A?):

1. *Algebra*: matrix factorizations and such

2. *Analysis*: perturbation theory, convergence, error analysis, etc.

3. *Algorithms*: efficient methods for modern machines.

4. *Applications*: coming from all over

The interplay between these aspects of the subject is a big part of what makes the area so interesting!

We could also organize around a few general techniques

1. Matrix factorizations

2. Perturbation theory and condition numbers

3. Floating point arithmetic

4. Complexity analysis

5. Software engineering

The point of the next few lectures will be to introduce these themes (4A) and (some of) our techniques in the context of two very simple operations: multiplication of a matrix by a vector, and multiplication of two matrices

I will assume that you have already had a course in linear algebra and that you know how to program. It will be helpful if you've had a previous numerical methods course, though maybe not strictly necessary; talk to me if you feel nervous on this point. The course will involve MATLAB programming, so it will also be helpful – but not strictly necessary – for you to have prior exposure to MATLAB. If you don't know MATLAB already but you have prior programming experience, you can learn MATLAB quickly enough. There may be some optional or extra credit problems that involve programming in another language, but nothing required.

Matrix algebra versus linear algebra

1. Matrices are extremely useful. So are linear transformations. But note that matrices and linear transformations are *different* things! Matrices *represent* finite-dimensional linear transformations with respect to particular bases. Change the bases, and you change the matrix, if not the underlying operator. Much of the class will be about finding the right

basis to make some property of the underlying transformation obvious, and about finding changes of basis that are "nice" for numerical work.

2. A linear transformation may correspond to different matrices depending on the choice of basis, but that doesn't mean the linear transformation is always the thing. For some applications, the matrix itself has meaning, and the associated linear operator is secondary. For example, if I look at an adjacency matrix for a graph, I probably really do care about the matrix – not just the linear transformation.

3. Sometimes, we can apply a linear transformation even when we don't have an explicit matrix. For example, suppose $F : \mathbb{R}^n \to \mathbb{R}^m$, and I want to compute $\partial F/\partial v|_{x_0} = (\nabla F(x_0)) \cdot v$. Even without an explicit matrix for ∇F, I can compute $\partial F/\partial v|_{x_0} \approx F(x_0 + hv) - F(x_0))/h$. There are many other linear transformations, too, for which it is more convenient to apply the transformations than to write down the matrix – using the FFT for the Fourier transform operator, for example, or fast multipole methods for relating charges to potentials in an n-body electrostatic interaction.

Matrix-vector multiply

Let us start with a very simple MATLAB program for matrix-vector multiplication:

```
function y = matvec1(A,x)
% Form y = A*x (version 1)

[m,n] = size(A);
y = zeros(m,1);
for i = 1:m
  for j = 1:n
    y(i) = y(i) + A(i,j)*x(j);
  end
end
```

We could just as well have switched the order of the i and j loops to give us a column-oriented rather than row-oriented version of the algorithm. Let's consider these two variants, written more compactly:

```
function y = matvec2_row(A,x)
% Form y = A*x (row-oriented)

[m,n] = size(A);
y = zeros(m,1);
for i = 1:m
  y(i) = A(i,:)*x;
end


function y = matvec2_col(A,x)
% Form y = A*x (column-oriented)

[m,n] = size(A);
y = zeros(m,1);
for j = 1:n
  y = y + A(:,j)*x(j);
end
```

It's not too surprising that the builtin matrix-vector multiply routine in MATLAB runs faster than either of our `matvec2` variants, but there are some other surprises lurking. Try timing each of these matrix-vector multiply methods for random square matrices of size 4095, 4096, and 4097, and see what happens. Note that you will want to run each code many times so that you don't get lots of measurement noise from finite timer granularity; for example, try

```
tic;            % Start timer
for i = 1:100 % Do enough trials that it takes some time
  % ...          Run experiment here
end
toc             % Stop timer
```

Basic matrix-matrix multiply

The classic algorithm to compute $C := C + AB$ is

```
for i = 1:m
```

```
    for j = 1:n
      for k = 1:p
        C(i,j) = C(i,j) + A(i,k)*B(k,j);
      end
    end
  end
```

This is sometimes called an *inner product* variant of the algorithm, because the innermost loop is computing a dot product between a row of A and a column of B. We can express this concisely in MATLAB as

```
  for i = 1:m
    for j = 1:n
      C(i,j) = C(i,j) + A(i,:)*B(:,j);
    end
  end
```

There are also *outer product* variants of the algorithm that put the loop over the index k on the outside, and thus computing C in terms of a sum of outer products:

```
  for k = 1:p
    C = C + A(:,k)*B(k,:);
  end
```

Blocking and performance

The basic matrix multiply outlined in the previous section will usually be at least an order of magnitude slower than a well-tuned matrix multiplication routine. There are several reasons for this lack of performance, but one of the most important is that the basic algorithm makes poor use of the *cache*. Modern chips can perform floating point arithmetic operations much more quickly than they can fetch data from memory; and the way that the basic algorithm is organized, we spend most of our time reading from memory rather than actually doing useful computations. Caches are organized to take advantage of *spatial locality*, or use of adjacent memory locations in a short period of program execution; and *temporal locality*, or re-use of the same memory location in a short period of program execution. The basic matrix multiply organizations don't do well with either of these. A better

organization would let us move some data into the cache and then do a lot of arithmetic with that data. The key idea behind this better organization is *blocking*.

When we looked at the inner product and outer product organizations in the previous sections, we really were thinking about partitioning A and B into rows and columns, respectively. For the inner product algorithm, we wrote A in terms of rows and B in terms of columns

$$\begin{bmatrix} a_{1,:} \\ a_{2,:} \\ \vdots \\ a_{m,:} \end{bmatrix} \begin{bmatrix} b_{:,1} & b_{:,2} & \cdots & b_{:,n} \end{bmatrix},$$

and for the outer product algorithm, we wrote A in terms of colums and B in terms of rows

$$\begin{bmatrix} a_{:,1} & a_{:,2} & \cdots & a_{:,p} \end{bmatrix} \begin{bmatrix} b_{1,:} \\ b_{2,:} \\ \vdots \\ b_{p,:} \end{bmatrix}.$$

More generally, though, we can think of writing A and B as *block matrices*:

$$A = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1,p_b} \\ A_{21} & A_{22} & \ldots & A_{2,p_b} \\ \vdots & \vdots & & \vdots \\ A_{m_b,1} & A_{m_b,2} & \ldots & A_{m_b,p_b} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} & \ldots & B_{1,p_b} \\ B_{21} & B_{22} & \ldots & B_{2,p_b} \\ \vdots & \vdots & & \vdots \\ B_{p_b,1} & B_{p_b,2} & \ldots & B_{p_b,n_b} \end{bmatrix}$$

where the matrices A_{ij} and B_{jk} are compatible for matrix multiplication. Then we we can write the submatrices of C in terms of the submatrices of A and B

$$C_{ij} = \sum_k A_{ij} B_{jk}.$$